A Realistic Simulation Model for Peer-to-Peer Storage Systems

Abdulhalim Dandoush, Sara Alouf, and Philippe Nain
INRIA Sophia Antipolis, B.P. 93
06902, Sophia Antipolis Cedex, France

Abstract

The peer-to-peer (P2P) paradigm have emerged as a cheap, scalable, self-repairing and fault-tolerant storage solution. This solution relies on erasure codes to generate additional redundant fragments of each "block of data" in order to increase the reliability and availability and overcome the churn. When the amount of unreachable fragments attains a predefined threshold, due to permanent departures or long disconnections of peers, a *recovery process* is initiated to compensate the missing fragments, requiring multiple fragments of data of a given "block" to be downloaded in parallel for an enhanced service. Recent modeling efforts that address the availability and the durability of data have assumed the recovery process to follow an exponential distribution, an assumption made mainly in the absence of studies characterizing the "real" distribution of the recovery process. This work aims at filling this gap and better understanding the behavior of these systems through simulation while taking into consideration the heterogeneity of peers, the underlying network topologies, the propagation delays and the transport protocol. To that end, the distributed storage protocol is implemented in the NS-2 network simulator. This paper describes a realistic simulation model that captures the behavior of P2P storage systems. We provide some experiments results that show how modeling the availability and durability can be impacted by the recovery times distribution which is impacted in turn by the characteristics of the the network and the context.

keywords — Network Simulator NS-2, peer-to-peer storage system, stochastic simulation, download time, recovery process

1 Introduction and Related work

The peer-to-peer (P2P) model has proved to be an alternative to the Client/Server model and a promising paradigm for Grid computing, file sharing, voice over IP, backup and storage applications. A major advantage of P2P systems is that peers can build a virtual overlay network on top of existing architecture and topology. Each peer receives/provides a service from/to other peers through the overlay network; examples of such a service are sharing the capacity of its central processing unit, sharing its bandwidth capacity, sharing its free storage space, and sharing local information about neighbors to help each other locating resources.

P2P storage systems (P2PSS) have emerged as a cheap, scalable, self-repairing and fault-tolerant solution. Such distributed systems rely on data fragmentation and distributed storage. Files are partitioned into fixed-size blocks that are themselves partitioned into fragments. Fragments of same block of data are usually stored on different peers. Given this configuration, a user wishing to retrieve a given data would need to perform multiple downloads, generally in parallel for an enhanced service, thereby saturating the download capacities of peers and reducing the download time. To mitigate churn of peers, redundancy mechanisms and a recovery process are needed. Redundant fragments are continuously injected in the system, thus maintaining data redundancy above a minimum desired level. When the amount of unreachable fragments attains a predefined threshold, a *recovery process* is initiated.

This paper considers systems relying on erasure codes to generate the redundant fragments. If s denotes the initial number of fragments of any block of data and r denotes the amount of additional redundant fragments generated using an erasure code algorithm (e.g. [21]) taking as input the s original fragments, then any s out of the $s+r$ fragments of a given block of data can be used to generate a new redundant fragment or to reconstruct the original block. Observe that this notation covers the case of replication-based systems, with $s = 1$ and r denoting the number of replicas. In fact, when a full-replication redundancy mechanism is used, the system will store r additional copies of each block of data over r different active peers in the network.

The recovery process includes the download of s fragments in order to generate one or more missing fragments of a given block of data, stored in the system initially as $s + r$ fragments (as the case of downloading the full block of data).

P2PSS may rely on a central authority that initiates the recovery process when necessary. This central authority could reconstruct all missing fragments of a given block of data and remotely store them on as many new peers. By "new" peers, we refer to peers that do not already store fragments of the same block of data. Alternatively, secure agents running on new peers could reconstruct by themselves missing fragments to be stored on the peers disks. A more detailed description of P2PSS, their recovery schemes and their policies is presented in Section 2.

Although the literature on modeling distributed systems, simulating P2P systems and parallel downloading is abundant, in particular the file sharing and resources allocation protocols (see [13, 23, 8]; non-exhaustive list), the P2PSS, in particular, the recovery process, is a subject that has not been analyzed.

K. Eger et al. [8] implemented a BitTorrent file sharing pro-

tocol in NS-2 and compared packet-level simulation results with flow-level for the download time of one file among an active peer-set. They showed that the propagation delay can significantly influences the download performance of BitTorrent.

V. Aggarwal et al. [1] implemented Gnutella file sharing protocol in SSFNet simulator and compared also the packet-level and the flow-level simulation results. They reflected the user behavior models in their simulation framework.

Q. He et al. [13] implemented a framework of P2P file sharing, in particular Gnutella, for NS-2, and they showed how Gnutella system performance can be impacted by the network characteristics.

1.1 Motivation

There have been recent modeling efforts focusing on the performance analysis of P2PSS in terms of data durability and data availability. In [20], Ramabhadran and Pasquale analyze systems that use *full replication* for data reliability. They develop a Markov chain analysis, then derive an expression for the lifetime of the replicated state and study the impact of bandwidth and storage limits on the system. This study relies on the assumption that the recovery process follows an exponential distribution. Observe that in replication-based systems, the recovery process lasts mainly for the download of one fragment of data that is equal to one block as the block here is not fragmented. In other words, the authors of [20] are implicitly assuming that the fragment download time is exponentially distributed.

In [2], we developed a more general model than that in [20], which applies to both replicated and erasure-coded P2PSS. Also, unlike [20], the model presented in [2] accounts for transient disconnections of peers, namely, the churn in the system. But we also assumed the recovery process to be exponentially distributed. However, this assumption can differ between replicated and erasure-coded P2PSS, as in the latter systems the recovery process is much more complex than in the former systems. Furthermore, the recovery process differs from centralized to distributed implementation.

In both studies, findings and conclusions rely on the assumption that the recovery process is exponentially distributed. However, this assumption is not supported by any experimental data. To the best of our knowledge, there has been no simulation study characterizing this process in real P2PSS.

It is thus essential to characterize the distribution of download and recovery processes in P2PSS. Evaluating these distributions is crucial to validate (or invalidate) some key assumptions made in the above works. Moreover, simulation is critical to the building and better understanding of these systems, in particular the availability and durability of data with the presence of realistic topologies, the underlying network protocols and peers characteristics.

The main objective of this paper is the description of the simulator itself, but we will show also in Section 4 through intensive simulations of many realistic scenarios that (i) the fragment download time follows closely an exponential distribution and (ii) fragment download times are weakly correlated in some interesting scenarios. Given that in erasure-coded systems, the block download time consists of downloading several fragments in parallel, it follows that the recovery process should follow approximately a hypo-exponential distribution of several phases. (This is nothing but the sum of several independent random variables exponentially distributed having each its own rate [12]). We found that this is indeed the case in some interesting contexts. We realized that beside the fact that the total workload is equally distributed over the active peers, there are two main reasons for the weak correlation between concurrent downloads as observed in some scenarios: (i) the good connectivity of the core network and (ii) the asymmetry in peers upstream and downstream bandwidths. So, as long as the bottleneck is the upstream capacity of peers, the fragment download times are close to be independent. Results of this simulator suggest that the models presented in [2] give accurate results on data durability and availability only in replicated P2PSS (as in [20]). The case of erasure-coded systems was inaccurately studied if the three above situations are met in the network.

Building on these results, we have incorporated into the model of [2] the assumption that fragment download and upload times (instead of the block download times or the recover times) are exponentially distributed with parameters α and β, respectively. The resulting models, appeared in [5], characterize data lifetime and availability in P2PSS storage systems that use either replication or erasure codes, under more realistic assumptions and well-known contexts as supported by results of the simulator presented in this paper and in [6].

1.2 Why do we use simulations and why can NS-2 be a good candidate?

To collect traces of fragment download/upload times, of block download times and of recovery times, one can choose to perform simulations or experimentations either on testbeds or on real networks. We would like to consider situations where peers are either homogeneous or heterogeneous, different underlying network topologies, and different propagation delays in the network. Also, we would like to consider systems with a large number of peers. To achieve all this with experiments over real networks is very difficult. Setting up experiments over a dedicated network like Planet-Lab [19] would require a long time, and there will be limitations on changing the topology and the peers characteristics. In addition, measurement-based studies do not allow to evaluate performance in advance of building and deploying the system, hence the importance of simulations at reasonable scale for the thorough evaluation of P2PSS before their deployment. We find it most attractive to implement the distributed storage protocol in a well-known network simulator and to simulate different scenarios. We choose NS-2 as network simulator because it is an open source discrete event simulator targeted at networking research. NS-2 provides substantial support for simulation of TCP and routing and it is well known and well validated.

Note that in view of the application specifications and ob-

jectives which are different from file sharing, grid computing or video streaming systems, involving some thousands (1–3) of peers in a simulation has to conclude realistic results and helps to understand the system behavior.

The rest of this paper is organized as follows. Section 2 overviews the storage protocol that we consider. Section 3 describes the simulation architecture, the methodology and the setup of the simulations. In Section 4, some experimental results are discussed. Last, Section 5 concludes the paper and highlights the future work.

2 System Description

We will describe in this section the storage protocol that we want to simulate:

- Files are partitioned into fixed-size blocks (the block size is S_B) that are themselves partitioned into s fragments (the fragment size is S_F).
- Each block is stored as a total of $s + r$ fragments, r of them are redundant and generated using erasure codes.
- Fixing block and fragment sizes helps to fix the value of the parameters s and r in the system for all stored blocks. These $s + r$ fragments are stored over $s + r$ different peers.
- Mainly for privacy issues, a peer can store at most one fragment of any block of data.
- Only the latest known location of each fragment is tracked, whether it is a connected or disconnected peer.
- To overcome churn and maintain data reliability and availability, unreachable fragments are continuously recovered.
- The number of connected peers at any time is typically much larger than the number of fragments associated with a block of data, i.e., $s + r$. Therefore, there are always at least $s + r$ *new* connected peers which are ready to receive and store fragments of a block of data.
- Once an unreachable fragment is recovered, any other copy of it that "reappears" in the system due to a peer reconnection is simply ignored, as only one location of the fragment (the newest one) is recorded in the system. Similarly, if a fragment is unreachable, the system knows of only one disconnected peer that stores the unreachable fragment.

Two implementations of the recovery process are considered. This process is triggered for each block whose number of unreachable fragments reaches a threshold k.

In the centralized implementation, a central authority will: (1) download *in parallel* s fragments from the peers which are connected, (2) reconstruct at once all unreachable fragments (by now considered as missing), and (3) upload them all *in parallel* onto as many new peers for storage. In fact, Step 2 executes in a negligible time compared to the execution time of Steps 1 and 3 and then will be neglected in the current simulator but can be added in the future. Step 1 (resp. Step 3) execution completes when the last fragment completes being downloaded (resp. uploaded).

In the distributed implementation, a secure agent on one new peer is notified of the identity of *one* out of the k unreachable fragments for it to reconstruct it. Upon notification, the secure agent (1) downloads s fragments from the peers which are connected to the storage system, (2) reconstructs the specified fragment and stores it on the peer's disk; (3) the secure agent then discards the s downloaded fragments so as to meet the privacy constraint that only one fragment of a block of data is held by a peer. This operation iterates until less than k fragments are sensed unreachable and stops if the number of missing fragments reaches $k - 1$. The recovery of one fragment lasts mainly for the execution time of Step 1; the recovery is simulated to be completed then as soon as the last fragment (out of s) completes being downloaded.

When $k = 1$, the recovery process is said to be *eager*; when $k \in \{2, \ldots, r\}$, the recovery process is said to be *lazy*.

3 Implementation Details

This section describes the base classes P2P_Storage_Directory, P2P_Storage_App, P2P_Storage_Wrapper and data structure. In fact, we follow the same methodology as the Web cash application presented in the NS Manual (cf. [9, Chap. 40]), and use some of the technical ideas presented in [9, Chap. 39,41] of the NS Manual, [8] and [4]. Therefore, we will discuss some selected pieces of code and sketch the description of the basic APIs, through which applications find data and request services from underlying transport NS agents. We implemented the P2PSS application in NS-2 (version 2.33) following the architecture depicted in Fig. 1 where the P2P_Storage_Wrapper object is an intermediate class that passes the data between the FullTcp transport agent object in NS-2 and the P2P_Storage_App class that represent the P2PSS application and takes care of crating the connections between applications.

Similarly to any simulation that uses NS, we define the basic parameters such as maximum number of stored files, maximum

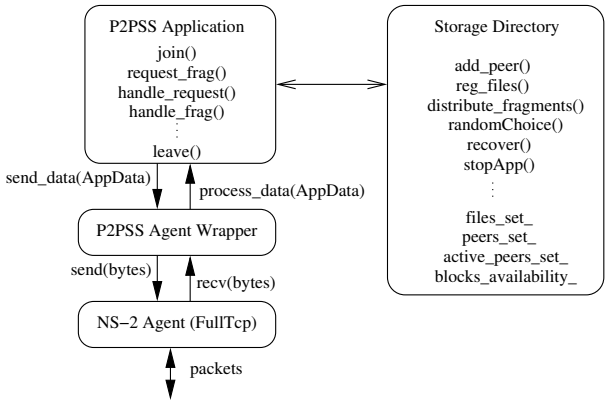


Figure 1: Simulator architecture.

number of peers, block size, fragment size, amount of redundancy and the recovery threshold at OTcl level in a TCL script as follows.

Listing 1: Simulation scenario setup

```
set NS [new Simulator]
#Number of peers
        set N_P 1000
        set Inter_arrival_req 3
#number of files
        set N_F 10000
        set max_request 1000
#overhead storage r/s
        set oh_st 1.5
#recovery threshold k
set k_th 1
#application type, e-library-like=1
# buckup-like type = 0
set app_type 1
#data unit sizes
        set S_F_KB [expr 16 *1024]
        set S_b_KB [expr 4 *1024]
        set S_Frag_KB [expr 1 *1024]
# set MSS for TCP
Agent/TCP/FullTcp set segsize_ 1460
#create instance of system directory
set dir [new P2P_Storage_Directory $N_P$ ...]
```

We instantiate then from P2P_Storage_Directory class the system directory object. The basic function members of the class P2P_Storage_Directory which is implemented as a child class of TclObject, as shown in Listing 2, are found in Table 1

Listing 2: Definition of the system directory class

```
class P2P_Storage_Directory : public TclObject
{

public:

   // The Constructor of the class
   P2P_Storage_Directory(int N_P,int N_F,...);
        ...
protected:
   // Tcl command interpreter
   int command(int ar,const char*const* argv);
        ...
};
```

To make it possible to create an instance of the system directory object in OTcl, we have to define a linkage object that must be derived from TclClass. This is illustrated in Listing 3. In fact, once NS is started, it executes the constructor for the static variable "class_p2p_storage_directory", and thus an instance of "P2P_Storage_DirectoryClass" is created.

Listing 3: The linkage object P2P_Storage_DirectoryClass between OTCL and C++ class P2P_Storage_Directory

```
static class P2P_Storage_DirectoryClass :
                                public TclClass
{
public:
      P2P_Storage_DirectoryClass() :
      TclClass("P2P_Storage_Directory") {}
      TclObject* create(int argc, ...)
         {
                if (argc != 10)
                        return NULL;
                else
                ...
         }
} class_p2p_storage_directory;
```

We assume that there is a given number of stored files in the system and before that peers request data, the system directory object distributes the $s + r$ fragments of each block of data of all files over $s + r$ peers chosen uniformly among all the registered peers in the system. This is the task of the member functions "distribute_fragments()" and "reg_file()", where the system directory has a private vector containing pointers to the meta-data of the stored files. Listings 4 and 5 depict the details of the meta-data (file structure) of any stored file and the member files_set_ respectively. In fact, a DHT system [15] does not choose randomly peers in the network to store the fragments of each block but the distribution of data depends on the identifier space, the identifier of each node (its position in the space) and the hash value of the block itself. However, it is proved in [15] that DHT makes the number of keys per node uniformly distributed with high probability. In other words, with high probability each node is responsible for $O(1/N)$ of the identifier space where N is the number of peers. It is proved as well in [23] that the cost of the lookup phase in Chord-like protocol grows as the logarithm of the number of nodes. As a result, and in view of the fact that we involve some hundreds to some thousands of nodes, we neglect the lookup cost with respect to the download or recovery times and we do not implement DHT to reduce the complexity. In other words, we use the same class of the system directory for both recovery process implementations and we assume that the system has a perfect knowledge of the state.

Listing 4: The file_list_entry data structure

```
typedef struct file_list_entry {
      int file_id;
      long file_size;
      long block_size;
      long frag_size;
      int N_blocks; // file_size/block_size
      int N_frags; // s
      int total_N_frags;//s+r

/* map between each block' id (key) and a
 list of s+r peer's id, on which the
 fragments are stored */
      map<int,int *> >block_node_id_list_;
      map<int,vector<Node*> >block_node_list_;
```

Table 1: The basic prototypes of P2P_Storage_Directory class

Method	Functionality
map ¡int,vector¡Node*¿ ¿ get_peer_list_(int file_ID)	gets a list of peers (s usually) for each block to download a specific file
void add_peer(Node* peer)	adds new peer to the directory
void reg_file(file_list_entry* file)	adds the file entry to the files_set_
void stopApps()	stops all the applications and frees the memory when the maximum simulation time or the maximum number of requests are reached
void del_peer(Node* peer)	deletes peer from active_Peer_set_ when leaving the system
void go_off(long id, Node * node_)	reduces the blocks availability, checks the recovery threshold, del_peer
void go_on(long id, Node * node_)	increases the blocks availability if not recovered, add_active_peer
int randomChoice(int min, int max)	chooses an active peer randomly
virtual void distribute_fragments()	distributes the fragments of blocks of the registered files
bool recovery(int block_id, int missing)	recovers missing fragments of a given block

```
    map<int,int> blocks_availability;
    map<int, bool> black_list;//false==lost
};
```

Listing 5: The private member files_set_ of the P2P_Storage_Directory class

```
vector<file_list_entry*> files_set_;
```

After creating the instance of the storage directory at the OTcl level, we allocate next the NS nodes and we create the underlying network topology by using for example the GT-ITM tool [3] (see more details in Section 3.1). We instantiate from P2P_Storage_App class the applications (peers) where a pointer at the Node class must be set to the attached application running on that Node (Agent) which will be used to pass data from an Agent to an Application. In fact, we did minor changes to the files: tcp-full.cc, tcp-full.h, node.cc, node.h, agent.cc and agent.h to support the collaboration between nodes, agents and the P2PSS applications.

Listing 6: OTcl level, creating nodes and application

```
set node($i) [$ns node]
        ...
set app($i) [new P2P_Storage_App $dir
        $node($i) $app_type $C_up($i)
        $Inter_arrival_req max_request]
        ...
```

We consider in fact two different storage applications, a backup-like application and an e-library-like application ("e" stands for "electronic"). In the first, a file stored in the system can be requested for retrieval only by the peer that has produced the file. In the second, any file can be downloaded by any peer in the system. In both applications, the storage protocol follows the description of Section 2.

Two types of requests are issued in the system. The first type is issued by the users of the system: a user issues a request to retrieve one of its files in the backup-like application, or a public document in the e-library-like application. The second type consists of management requests. Usually, these are issued by the central authority (in the centralized implementation of the recovery process) or by a peer (in the distributed implementation) as soon as the threshold k is reached for any stored block of data. In the simulator, these management requests are issued by the system directory object.

File download requests are translated into (i) a request to the directory service to obtain, for each block of the desired file, a list of at least s peers that store fragments of this block, (ii) opening TCP connections with each peer in the said list to download one fragment, (iii) registering some statistical information such as the start and the completion time of the downloaded data. All download requests issued by a given peer form a Poisson process. This assumption is met in real networks as found in [11]

Recovery requests are issued only in the scenarios where there is churn in the network. A recovery request concerning a given block translates into (i) a request from the directory service to a server in the centralized-repair scheme (we consider explicitly the first registered peer as the server in order to simulate the centralized implementation) or any active peer that is in charge of (ii) obtaining a list of at least s peers that store fragments of said block, (iii) opening TCP connections with each peer in the said list to download one fragment. Once all s fragments have been downloaded, the process proceeds with Steps 2 and 3, according to the implementation, as explained in Section 2. Last, the storage directory updates the system state at the end of the operation, namely it increases the availability level of the blocks of interest and points to the right locations of its fragments or otherwise it adds the lost block in a black list if the operation failed.

The P2PSS application uses many timers to handle events. In particular, a timer for scheduling the next file's request, a timer for scheduling the next failure moment once a peer becomes on line, and a timer for scheduling the next moment to rejoin the system once a peer becomes off line. We define the FileRequestTimer, OffLineTimer and OnLineTimer classes that are derived from the "TimerHandler" class, and write their "expire()" member functions to call file_request(), leave() and join() APIs respectively. Then, we included an instance of each timer object as a private

member of P2P_Storage_App object. Listings 7 and 8 show the example of FileRequestTimer and its expire member function implementation.

Listing 7: FileRequestTimer implementation

```
class FileRequestTimer:public TimerHandler
{
protected:
        P2P_Storage_App *app_;

public:
       FileRequestTimer(P2P_Storage_App* app):
                TimerHandler(), app_(app) {}
       inline virtual void expire(Event *);

};
```

Listing 8: Expire function of FileRequestTimer

```
void FileRequestTimer::expire(Event *) {
        app_->file_request();
}
```

Typically, applications access network services through sockets. NS-2 provides a set of well-defined API functions in the transport agent to simulate the behavior of the real sockets. Therefore, the P2P_Storage_Wrapper class handles calling the appropriate APIs when two applications want to communicate in order to (i) attache first the Full Tcp agent to both NS nodes via attach-agent and (ii) call then connect() instproc to set each agent's destination target to the other and last (iii) place one of them in LISTEN mode. We use in fact Full-Tcp agents since they support bidirectional data transfers.

Similar to what is done in the web cash application (see tcpapp.cc) we can model the underlying TCP connections as a FIFO byte stream, and then we will create same buffer management stuff. First, the P2P_Storage_Ms_Buf that contains a part of the messages such as the Request message and the Fragment message. Second, P2P_Storage_Msg_BufList implements a FIFO queues that will store all the sent messages (requests or data) on the sender side until they correctly and completely arrive to the destination side. In other words, there is no support in the class "Agent" to transmit different applications data and messages. Instead, as all data are delivered in sequence, we can view the TCP connections as a FIFO pipes, and the transfer of the application data will be emulated as follows. We first provide buffer for the application data at the sender to store the messages to be sent, next we use the Agent's API "sendmsg(int nbytes, const char *flags = 0)" to send a stream of an equivalent data size of the stored messages, then we count the bytes received at the destination. When the receiver has got all bytes of the current data or message transmission (first message in the FIFO on the sender side toward the receiver), then the receiver gets the data directly from the FIFO's sender. These are the tasks of the functions "send_data()" and "send()" on the sender application side and "process_data()" and "recv()" on the received side as shown in Fig.1 and described in Table 3, which use in turn the prototypes of the FIFO queues depicted in Table 2, where a FIFO queue is represented by the P2P_Storage_Msg_BufList class.

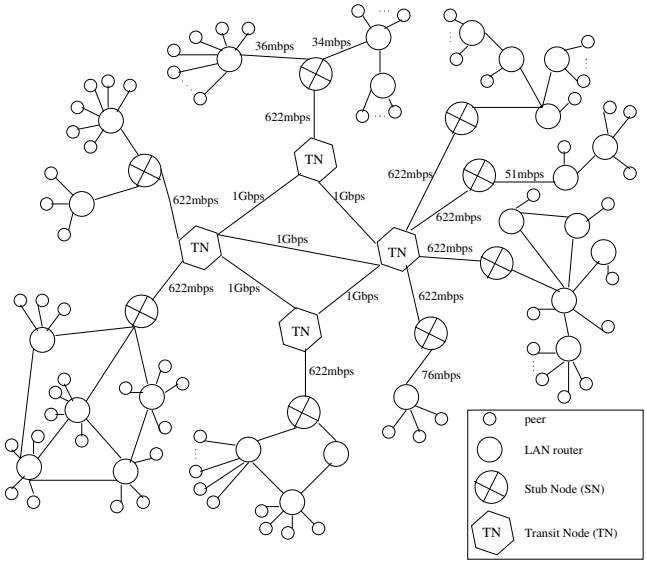


Figure 2: Three-level hierarchical random graph of Experiment 1.

3.1 Network Topology

Having a representative view of enterprise networks or the Internet topology is very important for a simulator to predict the behavior of a network protocol or application if it were to be deployed. In fact, the simulated topology often influences the outcome of the simulations. Realistic topologies are thus needed to produce realistic simulation results. Most of existing simulation studies have used representations of a real topology (e.g. the Arpanet), simple models (e.g. a star topology), or random flat graphs (i.e. non-hierarchical) that are generated by Waxman's edge-probability function [24].

However, random models offer very little control over the structure of the resulting topologies. In particular, they do not capture the hierarchy that is present in the Internet. Recently, tools such as BRITE [17] and GT-ITM [3] have been designed to generate more complex random graphs, that are hierarchical, to better approximate the Internet's hierarchical structure.

To produce realistic topologies for our simulations, we use the tool GT-ITM [3] to generate a total of six random graphs. Three levels of hierarchy are used corresponding to transit domains, stub domains, and local area networks (LANs) attached to stub domains. Each graph has one transit domain of four nodes; each of the nodes is connected to two or three other transit nodes. Each transit node is connected on average to two stub nodes, and each stub node is in turn connected on average to four routers. Behind every router there is a certain number of fully-connected peers constituting a LAN. The first of these six graphs is depicted in Fig. 2, where we have used the notation TN for "transit node" and SN for "stub node".

Table 2: The basic prototypes of P2P_Storage_Msg_BufList class

Method	Functionality
void insert(P2P_Storage_Msg_Buf *d)	stores msgs of the sender until the reception of their acks
P2P_Storage_Msg_Buf* detach()	if the data is received by the destination, deletes them from the FIFO buffer
int size()	returns the current size of the buffer

3.2 Experiments Setup

We will present results of four experiments that represent different contexts. Experiments 1 and 2 used the random graphs generated with the GT-ITM tool as detailed earlier, whereas a simple star topology is used in Experiments 3 and 4. Regarding the intra- and inter-domain capacities, we rely, in the first experiment, on the information provided by RENATER [22] and GÉANT [10] web sites. In those networks, the links are well-provisioned. To have a more complete study, we consider, in Experiment 2, links with smaller capacities, as can be seen in rows 4–6 of Table 4. Propagation delays over TN-SN edges vary from edge to edge as can be seen in row 7 of Table 4. Let C_u and C_d denote respectively the upload and download capacity of a peer. To set these values, we rely mainly on the findings of [11] and [14]. The experimental study of file sharing systems and of the Skype P2P voice over IP system [11] found that more than 90% of users have upload capacity C_u between 30Kbps and 384Kbps. However, the measurement study [14] done on BitTorrent clients in 2007 reports that 70% of peers have an upload capacity C_u between 350Kbps and 1Mbps and even 10% of peers have an upload capacity between 10Mbps and 110 Mbps. The capacities that we have selected in the simulations vary uniformly between the values of the ISDN and ADSL technologies; they can be found in rows 8–9 of Table 4. Observe that, except in Experiment 4, peers are heterogeneous. In addition, we consider a symmetric upload/download peer's capacities in Experiment 4 with $C_u = C_d = 384$kbps. We will attribute the propagation delays over routers-peers edges randomly between 1ms and 25ms, except in Experiment 4 where a fixed delay of 2ms is considered as can be seen in row 10 of Table 4.

In Experiments 1 and 2, there exists a background traffic between three pairs of routers across the common backbone. This traffic consists of random exponential and CBR traffic over UDP protocol and FTP traffic over TCP.

In each of the experiments, the amount of data transferred between routers and peers in the system during the observed time (that is from 4e+5 up to 5e+6 seconds) are, on average, 4.5–9 GB of P2P application traffic, and when applicable 150–350 MB of FTP, 200–400 MB of CBR, and 250–500 MB of the exponential traffic. In each of the experiments, the P2P traffic is well distributed over the active peers.

Experiment 2 simulates a backup-like application whereas the other experiments simulate an e-library-like application. Churn is considered only in Experiments 2 and 3. As a consequence, redundancy is added and maintained only in these experiments. The storage overhead r/s is 1 and 0.5 respectively. We consider the distributed implementation of the recovery process in Experiment 2, and the centralized implementation of the same in Experiment 3; the eager policy ($k = 1$) is considered in both experiments. In other words, once a peer disconnects from the system, all fragments that are stored on it must be recovered.

Churn is implemented as follows. We assume in the simulations that the successive on-times (respectively off-times) of a peer are independent and identically distributed random variables with a common exponential distribution function with parameter $\mu_1 > 0$ (respectively $\mu_2 > 0$). This assumption is in agreement with the analysis in [20]. We consider $1/\mu_1 = 3$ hours and $1/\mu_2 = 1$ hours.

Download requests are generated at each peer according to a Poisson process. This assumption is met in real networks as found in [11]. We assume all peers have the same request generation rate, denoted λ. We vary the value of λ across the experiments as reported in row 16 of Table 4.

The last setting concerns the files that are stored in the P2PSS. Fragment sizes S_F (resp. block sizes S_B) in P2P systems are typically between 64KB and 4MB each (resp. between 4MB and 9MB each). We will consider in most of our experiments $S_F = 1$MB except in Experiment 2 where $S_F = 512$KB. We consider $S_B = 8$MB in Experiments 1 and 3, and $S_B = 4$MB in Experiments 2 and 4. Therefore $s = 4$ or 8. As for the file size, we assume for now that it is equal to the block size. Therefore, the file download size is actually the block download size. Observe that the recovery process is related to the block download time and not to the file download time.

Table 4 summarizes the key settings of the experiments.

4 Experimental Results

In this section, we present the results of our simulations and the inference that we can draw from them. For each experiment, we collect the fragment download time, the block download time and the recovery time when applicable. In Experiment 2 (distributed recovery), the two latter durations are collected to the same dataset as there is no essential difference between them. Having collected these samples, we compute the sample average and use MLE, LSE and EM algorithms to fit the empirical distributions. Concerning the fragment download time, we perform the Kolmogorov-Smirnov test [16] on the fitted distribution. In the following, we will present selected results from Experiments 1, 3 and 4. The results of the second experiment are briefly reported in Tables 5–6.

Table 3: The basic prototypes of P2P_Storage_App and P2P_Storage_Wrapper classes

Method	Functionality
virtual void start()	after calling the constructor, App starts requesting files with an inter-request times chosen from an exponential distribution
double exponential(double lambda)	generates a random number from an exponential distribution
int create_conn(Node *dst,int file_id,int block_id, int frag_id)	establishes a connection with the destination
virtual void send_data(P2P_Storage_Msg m, int s_id, int dst_id)	Application sends msg to the wrapper agent
virtual void send(int nbytes)	wrapper agent calls sendmsg() of the tcp agent
void recv(int nbytes, int socket_id)	the NS agent announces the App each time a packet arrives
void process_data(P2P_Storage_Msg msg, int s_id)	handles the received data
void close_connections(int conn_id)	requests the agent to close the connections after completing a download if no other data to be sent or to be received
void file_request(int file_id)	creates connections and sends requests to get the file after calling the Directory member function get_peer_list_(int file_id)
void request_frag(int conn_id, int f_id, b_id, int fr_id, int dst_id)	requests a fragment from a peer
void handle_request(P2P_Storage_Msg m, int conn_id)	handles a request, creates a fragment message and sends it
void handle_frag(P2P_Storage_Msg m, int conn_id)	called when the receiver gets all bytes of the current transmission, updates the related members, increases the number of completed fragments, calls reg_frag_traces(), and frag_downloaded()
void reg_frag_traces(int file_id, int b_id, int fr_id, int dst_id)	registers the information about a completely received fragment
void frag_downloaded(int f, int b, int frag, int conn,int dst)	after completing a download of fragment, calls close_connections(), removes the uploader from the map container between block_id and uploader_id
void join()	scheduled by the OnLineTimer, initializes the instance of FileRequestTimer and the OffLineTimer, informs the directory that the peer is active, and calls go_on() function of the directory class
void leave()	scheduled by the OffLineTimer, closes all the connections, cleans the memory, initializes the instance of OnLineTimer, informs the directory about the failure and calls go_off() function of the directory class
int randomChoice(int min, int max)	chooses a file to be requested

4.1 Experiment 1

We have collected 76331 samples of the fragment download time (cf. column 2 of Table 5). The empirical cumulative distribution function (CDF) is depicted in Fig. 3(a). We can see that it is remarkably close to the exponential distribution. Two exponential distributions are plotted in Fig. 3(a), each having a different parameter, derived from a different fitting technique. The two techniques that we used are MLE and LSE. The parameter returned by MLE is nothing but the inverse of the sample average and is denoted α; see row 2 of Table 5.

Beyond the graphical match between the empirical distribution and the exponential distribution, we did a hypothesis test. Let $\mathbf{X}$ be a vector storing the collected fragment download times. The Kolmogorov-Smirnov test compares the vector $\mathbf{X}$ with a CDF function, denoted cdf (in the present case, it is the exponential distribution), to determine if the sample $\mathbf{X}$ could have the hypothesized continuous distribution cdf. The null hypothesis is that $\mathbf{X}$ has the distribution defined in cdf, the alternative one being that $\mathbf{X}$ does not have that distribution. We reject the null hypothesis if the test is significant at the $l\%$ level. In Experiment 1, the null hypothesis with $\alpha = 1/40.35$ is not rejected for $l = 7\%$.

Looking now at concurrent downloads, we have found that these are weakly correlated and close to be independent. Beside the fact that the total workload is equally distributed over the active peers, there are two main reasons for the weak correlation between concurrent downloads as observed in Experiment 1: (i) the good connectivity of the core network and (ii) the asymmetry in peers upstream and downstream bandwidths. So, as long as the bottleneck is the upstream capacity of peers, the fragment download times are close to be independent.

Regarding the block download times, we have collected 9197 samples. The sample average is given in row 7 of Table 5. The empirical CDF is plotted in Fig. 3(b). We followed the same methodology and computed the closest exponential distribution using MLE. However, the match between the two distribution appears to be poor, and actually, the alternative hypothesis is not rejected in this case.

To find a distribution that will more likely fit the empirical data, we make the following analysis. To get a block of data, s fragments, stored on s different peers, have to be downloaded. This is

Table 4: Experiments setup

Experiment number	1	2	3	4
Topology	random	random	star	star
Number of peers	960	640	480	250
TN-TN capacities (Gbps)	1	1	—	—
TN-SN capacities (Mbps)	622	10–34	—	—
SN-routers capacities (Mbps)	34–155	4–10	—	—
TN-SN delays (ms)	5–25	5–25	—	—
C_u of peers (Kbps)	150–1000	256–700	256–700	384
C_d of peers (Kbps)	$8 \times C_u$	$10 \times C_u$	2048	384
routers-peers delays (ms)	1–20	1–10	1–25	2
Background traffic	yes	yes	no	no
Application type	e-library	backup	e-library	e-library
Peers churn	no	yes	yes	no
Recovery process	—	distributed	centralized	—
r	—	s	$s/2$	—
$1/\lambda$ (min.)	80	160	16	1/60
S_B (MB)	8	4	8	4
S_F (KB)	1024	512	1024	1024
s	8	8	8	4

more efficiently done in parallel and this is how we implemented it in the simulator. We have seen that the download of a single fragment is well-modeled by an exponential random variable with parameter α. Also, concurrent downloads were found to be close to independent. Therefore, the time needed for downloading s fragments in parallel is distributed like the maximum of s "independent" exponential random variables, which, due to the memoryless property (see also [12]), is the sum of s independent exponential random variables with parameters $s\alpha, (s-1)\alpha, \ldots, \alpha$. This distribution is called the hypo-exponential distribution and its expectation is

$$E[T] = 1/\alpha \sum_{i=1}^{s} 1/i \qquad (1)$$

where T denotes the block download time (or equivalently the distributed recovery duration).

In Experiment 1, $E[T] = 109.66$ seconds, while the sample average is equal to 102.75; cf. column 2 of Table 6. The relative error is 6.7%. The hypo-exponential distribution with s phases and parameters $s\alpha, (s-1)\alpha, \ldots, \alpha$ is plotted in Fig. 3(b). This distribution has a very good visual match with the empirical CDF of the block download time.

As a next step, we apply an EM algorithm [7] to find the best hypo-exponential distribution with s phases that fits the empirical data. In particular, we use EMpht [18], which is a program for fitting phase-type distributions to collected data. We do not plot the outcome of this program in Fig. 3(b) as it mainly overlaps with the hypo-exponential distribution with s phases and parameters $s\alpha, (s-1)\alpha, \ldots, \alpha$ that is already plotted there. After performing the Kolmogorov-Smirnov test, we find that the null hypothesis is not rejected for $l = 7\%$ (same significance level as for the fragment download times).

We conclude the analysis of the first experiment's results with four important points:

- The exponential assumption on the block download time is not met in realistic simulations.
- The fragment download time could be modeled by an exponential distribution with parameter α equal to the inverse of its average.
- Download times are weakly correlated and close to be independent as long as the bottleneck is the upstream capacity of peers.
- As a consequence, the block download time could be modeled by a hypo-exponential distribution with s phases and parameters $s\alpha, (s-1)\alpha, \ldots, \alpha$.

  However, the null hypothesis for the block download time or the recovery process, that it follows hypo-exponential distribution, is not always rejected. This is the case of Experiments 3 and 4, as seen next.

4.2 Experiment 3

In this experiment, peers are not always connected. Each time a peer disconnects from the network, all the fragments that were stored on his disk will have to be recovered. The recovery process is implemented in a centralized way.

The empirical CDF of the fragment download time and that of the recovery time are reported in Fig. 4. Following the same methodology as that used to analyze the results of Experiment 1, we find that the alternative hypothesis on the recovery process distribution is not rejected in spite of the fact that the system

Table 5: Summary of experiments results

Experiment number	1	2	3	4
Average frag. down. time $= 1/\alpha$ (sec.)	40.35	34.7367	40.722	135.867
Samples number	76331	9737	4669	37200
t_m (sec.)	8.77	6.84	16.4	25.377
$1/\hat{\alpha}$ (sec.)	39.351	32.106	32.05	110.49
$1/\beta$, $1/\hat{\beta}$ (sec.)	—	—	6.22, 5.11	—
Average of recovery or block down. time (sec.)	102.75	92.4762	89.848	205.19
Samples number	9197	589	561	9300

Table 6: Block download time or recovery process: Validation of the approximations introduced in Eqs. (1)–(3)

Experiment number	1	2	3	4
Sample average	102.75	92.48	89.85	205.19
Inferred average from Eqs. (1), (2)	109.66	94.40	116.89	283.05
Relative error (%)	6.7	2.1	30.1	37.95
Inferred average from Eqs. (4), (3)	106.95	94.10	92.21	255.564
Relative error (%)	4.1	1.8	2.6	24.55

workload is small and the bottleneck is the upstream capacity of peers. The relevant parameters are reported in column 4 of Tables 5 and 6.

There is a simple reason for that. We actually know that the download of a single fragment cannot be infinitely small, as suggested by the exponential distribution. Let t_m be the duration of the fastest fragment download among all s downloads. All other (slower) downloads are necessarily bounded by t_m. The effect of this minimum value can be neglected as long as t_m is negligible with respect to the average fragment download time. Otherwise, we need to consider that the fragment download/upload time is composed of two components: (i) a (constant) minimum delay t_m and (ii) a random variable distributed exponentially with parameter $\hat{\alpha}$ (resp. $\hat{\beta}$). This random variable models the collected data, shifted left by the value of t_m. The minimum delay can be approximated as $RTT + (S_F + \text{ Headers})/\max\{C_u\}$, where RTT stands for round-trip time.

The value of t_m is clearly visible in Fig. 4(a). We plot in this figure the empirical CDF of the fragment download time and the MLE exponential fits to both the collected and shifted data. The null hypothesis is rejected for the collected data but not rejected for the shifted data.

This is the same case of the recovery process of Experiment 2. Repeating the same analysis than in Section 4.1, and assuming that the fragment upload time follows an exponential distribution with parameter β, then the centralized recovery process, denoted T_c, would be modeled by a hypo-exponential distribution with $s + k$ phases ($k = 1$ in Experiment 3) having expectation

$$E[T_c] = 1/\alpha \sum_{i=1}^{s} 1/i + 1/\beta \sum_{j=1}^{k} 1/j \,. \tag{2}$$

Considering this distribution, we find that the null hypothesis of the Kolmogorov-Smirnov test for the collected data with parameters $1/\alpha = 40.72$ and $1/\beta = 6.22$ is rejected[1] for $l = 6\%$, while it is not rejected for the shifted data with parameters $1/\hat{\alpha} = 32.05$ and $1/\hat{\beta} = 5.11$.

Equations (1) and (2) should then be replaced with

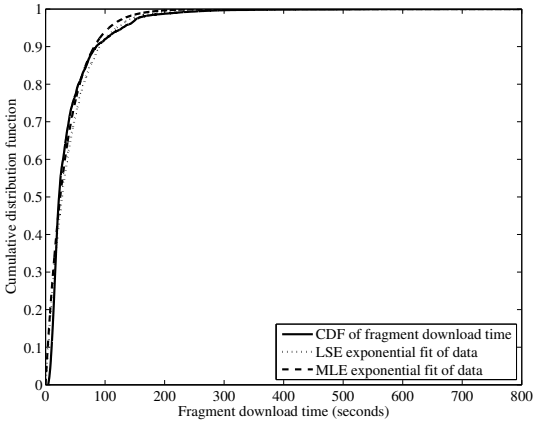
$$E[T] = t_m + 1/\hat{\alpha} \sum_{i=1}^{s} 1/i \,, \tag{3}$$

$$E[T_c] = t_m + 1/\hat{\alpha} \sum_{i=1}^{s} 1/i + 1/\hat{\beta} \sum_{j=1}^{k} 1/j \,. \tag{4}$$

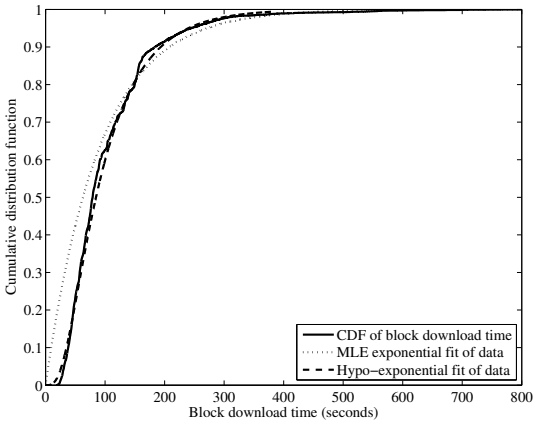
The averages inferred from Eqs. (1)–(4) are listed in rows 3 and 5 of Table 6, and their relative errors with respect to the sample average are listed in rows 4 and 6 of the same table. Observe that the inferred average improves across Experiments 1–3 when considering shifted data. The best improvement seen is that in Experiment 3. By considering that the *shifted* recovery time is hypo-exponentially distributed with $s + 1$ phases and parameters $s\hat{\alpha}, (s-1)\hat{\alpha}, \ldots, \hat{\alpha}, \hat{\beta}$, the relative error on the inferred average drops from 30.1% to 2.6%.

The conclusion of this discussion is that the exponential assumption on fragments download/upload time is met in most cases as long as the system workload is small and peers's download/upload capacities are asymmetric in such a way that the bottleneck is the upstream capacity of peers. The same exponential assumption does not hold on the block download time. The block download time and the recovery time are well approximated in Experiments 1,2 and 3-like scenarios by a hypo-exponential distribution.

[1] Even though it is rejected, this distribution is still much closer to the empirical data than the exponential distribution.

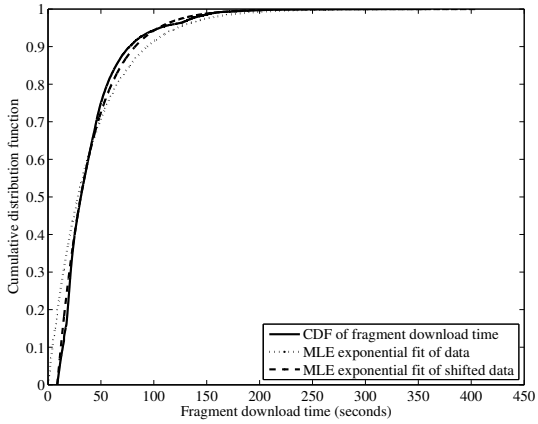


(a) Exponential fit of the fragment download time distribution

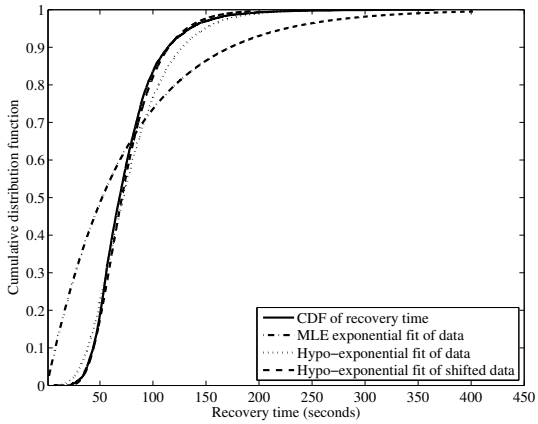


(b) Fitting of the block download time distribution

Figure 3: Experiment 1: Fragment and block download times.



(a) Exponential fit of the fragment download time distribution ignoring the minimum value



(b) Fitting of the recovery time distribution

Figure 4: Experiment 4: Fragment and recovery time, centralized recovery.

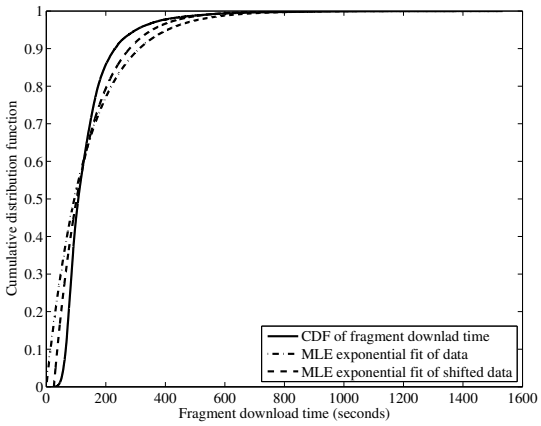
4.3 Experiment 4

In this experiment, peers are homogeneous and always connected. The system workload is relatively big and peers's download/upload capacities are symmetric in such a way that the bottleneck can be the upstream or the downstream capacity of peers. The concurrent fragment download processes are not independent but correlated. We see from Fig. 5(a) that the fragment download time is remarkably not exponentially distributed in such a scenario, even for the *shifted*, but it follows a phase type distribution unlike the case of Experiments 1–3.

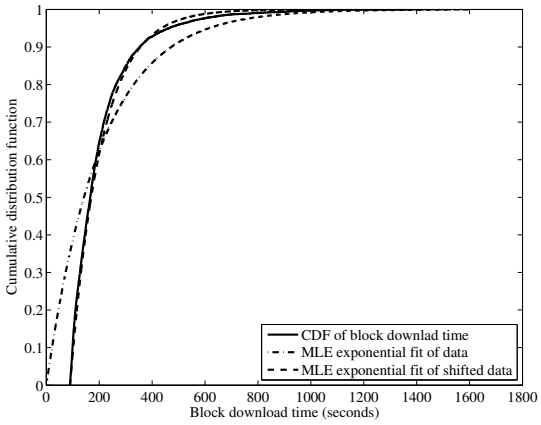
Regarding the block download time, we plot in Fig. 5 the empirical CDF of the data download time and the MLE exponential fits to both the collected and shifted data. It is remarkabl that the exponential distribution fits very well the data distribution. In fact, the null hypothesis is rejected for the collected data but not rejected for the shifted data.

5 Conclusion

This paper describes a realistic simulation model of the P2P storage system and sketches its implementation on top of the Network Simulator NS (version 2.33). It provides some simulation analysis of download and recovery processes in P2PSS. We set up four simulations which enable us to collect fragment/block download times and recoveries times under a variety of conditions. We show that the exponential assumption on the block download time can hold in some contexts such as the Experiment 4 context. The same assumption on fragments download/upload time is met in many cases implying that both the block download time and the recovery process could be modeled by a hypo-exponential distribution with a pre-determined number of phases. As a result, our simulations suggest that the models presented in [2] give accurate results on data durability and availability only in replicated P2PSS (as in [20]) or in Experiment 4 like-context. This was the motivation to realize the work in [5] where the models evaluate data durability and availability in more wide contexts.



(a) Exponential fit of the fragment download time distribution



(b) Fitting of the recovery or block download time distribution

Figure 5: Experiment 5: Fragment and block download times.

References

[1] V. Aggarwal, O. Akonjang, A. Feldmann, R. Tashev, and S. Mohrs. Reflecting p2p user behaviour models in a simulation environment. In *Proc. of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 16–523, Washington, DC, USA, 2008.

[2] S. Alouf, A. Dandoush, and P. Nain. Performance analysis of peer-to-peer storage systems. In *Proc. of 20th ITC*, volume 4516 of *LNCS*, pages 642–653, Ottawa, Canada, June 2007.

[3] K. Calvert, M. Doar, and E. W. Zegura. Modeling Internet topology. *IEEE Communications Magazine*, June 1997.

[4] J. Chung and M. Claypool. Ns by Example. http://nile.wpi.edu/NS/.

[5] A. Dandoush, S. Alouf, and P. Nain. Performance analysis of centralized versus distributed recovery schemes in P2P storage systems. In *Proc. of IFIP/TC6 NETWORKING 2009*, volume 5550 of *LNCS*, pages 676–689, Aachen, Germany, May 11–15 2009.

[6] A. Dandoush, S. Alouf, and P. Nain. Simulation analysis of download and recovery processes in p2p storage systems. In *Proc. of 21th ITC*, Paris, France, September 15–17 2009.

[7] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *J. Royal Statist. Soc.*, 39(1):1–37, 1977.

[8] K. Eger, T. Hobfeld, A. Binzenhofer, and G. Kunzmann. Efficient simulation of large-scale P2P networks: Packet-level vs. flow-level simulations. In *Proc. of UPGRADE-CN'07*, Monterey, California, USA, June 2007.

[9] K. Fall and K. Varadhan. The NS manual, the VINT project, UC Berkeley, LBL, USC/ISI, and Xerox PARC. http://www.isi.edu/nsnam/ns/ns-documentation.html, November 2008.

[10] GÉANT: a pan-European backbone which connects Europe's national research and education networks. http://www.geant.net/server/show/nav.159.

[11] A. Guha, N. Daswani, and R. Jain. An experimental study of the skype peer-to-peer VoIP system. In *Proc. of 5th IPTPS*, Santa Barbara, California, February 2006.

[12] P. Harrison and S. Zertal. Queueing models of RAID systems with maxima of waiting times. *Performance Evaluation Journal*, 64(7-8):664–689, August 2007.

[13] Q. He, M. Ammar, G. Riley, H. Raj, and R. Fujimoto. Mapping peer behavior to packet-level details: A framework for packet-level simulation of peer-to-peer systems. In *proc. of 11th IEEE/ACM MASCOTS*, Orlando, Florida, USA, October 2003.

[14] T. Isdal, M. Piatek, A. Krishnamurthy, and T. Anderson. Leveraging Bittorrent for end host measurements. In *Proc. 8th Passive and Active Measurement Conference*, Louvain-la-neuve, Belgium, April 2007.

[15] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM.

[16] F. J. Massey. The kolmogorov-smirnov test for goodness of fit. *J. Am. Statist. Assoc.*, 46(253):68—-78, 1951.

[17] A. Medina, A. Lakhina, I. Matta, and J. Byers. Brite: Boston University representative Internet topology generator. http://www.cs.bu.edu/brite/.

[18] M. Olsson. The EMpht-programme. Technical report, Department of Mathematics, Chalmers University of Technology, 1998.

[19] PlanetLab. An open platform for developing, deploying, and accessing planetary-scale services. http://www.planet-lab.org/, 2007.

[20] S. Ramabhadran and J. Pasquale. Analysis of long-running replicated systems. In *Proc. of IEEE Infocom '06*, Barcelona, Spain, April 2006.

[21] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *J. SIAM*, 8(2):300–304, June 1960.

[22] Renater: Le Réseau National de télécommunications pour la Technologie, l'Enseignement et la Recherche. http://www.renater.fr.

[23] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-to-Peer lookup service for Internet applications. In *Proc. of ACM Sigcomm '01*, pages 149–160, San Diego, California, August 2001.

[24] B. M. Waxman. outing of multipoint connections. *IEEE Journal on Selected Areas in Communications*, 6(9):1617–1622, 1988.